AlloyFL: A Fault Localization Framework for Alloy

Tanvir Ahmed Khan
University of Texas at Arlington
Arlington, USA
txk6771@mavs.uta.edu

Allison Sullivan
University of Texas at Arlington
Arlington,USA
allison.sullivan@uta.edu

Kaiyuan Wang
Google, Inc.
Sunnyvale, USA
kaiyuanw@google.com

ABSTRACT

Declarative models help improve the reliability of software systems: models can be used to convey requirements, analyze system designs and verify implementation properties. Alloy is a commonly used modeling language. A key strength of Alloy is the Analyzer, Alloy's integrated development environment (IDE), which allows users to write and execute models by leveraging a fully automatic SAT based analysis engine. Unfortunately, writing correct constraints of complex properties is difficult.To help users identify fault locations, AlloyFL is a fault localization technique that takes as input a faulty Alloy model and a fault-revealing test suite. As output, AlloyFL returns a ranked list of locations from most to least suspicious. This paper describes our Java implementation of AlloyFL as an extension to the Analyzer. Our experimental results show AlloyFL is capable of detecting the location of real world faults and works in the presence of multiple faulty locations. The demo video for AlloyFL can be found at https://youtu.be/ZwgP58Nsbx8.

CCS CONCEPTS

• **Software and its engineering → Software defect analysis**.

KEYWORDS

Alloy, Fault localization, Declarative programming

ACM Reference Format:
Tanvir Ahmed Khan, Allison Sullivan, and Kaiyuan Wang. 2021. AlloyFL: A Fault Localization Framework for Alloy. In *Proceedings of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '21), August 23–28, 2021, Athens, Greece.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3468264.3473116

1 INTRODUCTION

In today's society, we are becoming increasingly dependent on software systems. At the same time, we also constantly witness the negative impacts of buggy software. One way to help develop better software systems is to leverage software models, which can have numerous benefits throughout the software development lifecycle. Before systems are built, models can be used to automatically ensure design-level properties are satisfied [4, 7, 19]. After systems are built, models can be used to automatically test and verify implementations [11]. Alloy is a declarative, first-order logic modeling language that has been used to verify system designs in multiple domains, including security [8, 13], networking [15], and UML analysis [9, 10]. A key strength of Alloy is the Analyzer, an integrated development environment (IDE) for Alloy. The Analyzer allows users to write Alloy models and execute commands to exercise the model's constraints. To execute commands, the Analyzer performs a fully automated analysis using off the shelf SAT solvers to generate assignments to the sets and relations of the models such that a user specified property is satisfied.

The many benefits of software models can only be achieved if the model itself is correct. Fortunately, prior work has introduced AUnit, a unit testing framework for Alloy, which is designed to give users a systematic method to check if their model matches their expectations [18]. However, if an AUnit test suite reveals a model to be buggy, the user still needs to be able to localize and fix the faulty portion of the model. Alloy's expressive operators (e.g. transitive closure, quantified formulas) allow users to write succinct formulations of complex properties. Unfortunately, this same succinct representation makes localizing faulty Alloy constraints difficult. In the base version of the Analyzer, the only avenue users have to localize faults is the unsat core highlighting. However, Daniel Jackson, the inventor of Alloy, has acknowledged that the unsat core is insufficient [5] and the unsat core only helps when a constraint is unexpectedly unsatisfiable. To address this, our prior work developed AlloyFL, a fault localization technique for Alloy which adapts spectrum-based fault localization (SBFL) and mutation-based fault localization (MBFL) techniques designed for imperative languages to Alloy's declarative execution environment [21].

This paper describes our efforts to create a Java implementation of AlloyFL as an extension to the Analyzer's standalone executable jar (https://alloyfl.github.io). By extending the Analyzer, we are able to give users access to AlloyFL within the existing development workflow for Alloy models. Specifically, given a faulty model and a fault revealing AUnit test suite, AlloyFL returns a ranked list of suspicious abstract syntax tree (AST) node locations in the faulty model. AlloyFL conveys this information to the user by both updating the logging interface of the Analyzer to display the ranked list and highlighting locations in the text editor based on their suspiciousness score. Our implementation supports AlloyFL$_{hy}$, a hybrid fault localization technique which combines AlloyFL$_{co}$, a SBFL technique and AlloyFL$_{mu}$, a MBFL technique to utilize their individual strengths for detecting different types of faults in Alloy's declarative execution environment. To combine the techniques, AlloyFL$_{hy}$ computes a score for each AST node using a formula that aggregates AlloyFL$_{co}$'s score and AlloyFL$_{mu}$'s score.

2 BACKGROUND

In this section, we present a faulty Alloy model to introduce key concepts of Alloy, AUnit and AlloyFL.

ESEC/FSE '21, August 23–28, 2021, Athens, Greece

ACM ISBN 978-1-4503-8562-6/21/08.
https://doi.org/10.1145/3468264.3473116

```
1. sig List { header: lone Node }
2. sig Node { link: lone Node }
3. pred Acyclic(l: List) {
4.   all n: Node | n in l.header.*link => n !in n.*link }
5. run Acyclic
```

Figure 1: Acyclic Singly-linked List

```
1. val ValidListSizeTwo {
2.  some disj List0: List { some disj Node0, Node1: Node {
3.    List = List0 and Node = Node0 + Node1
4.    header = List0->Node1 and link = Node1->Node0
5.    @cmd: Acyclic[List0] } } }
6. @Test:  run ValidListSizeTwo expect 1
```

Figure 2: Example AUnit Test for List

Figure 1 displays a faulty model of a singly-linked list. Signature paragraphs introduce named sets of atoms and their relations into the model (lines 1 - 2). Line 1 introduces `List` as named set and line 2 introduces `Node` as a named set. The relation `header` declares that each `List` atom points to zero or one header node. Similarly, the relation `link` conveys that each `Node` atom points to zero or one subsequent node. Predicate paragraphs introduce named first-order logic formulas that can be invoked elsewhere (lines 3 - 4). The predicate `Acyclic` uses universal quantification (`all`) and reflexive transitive closure (*) to *incorrectly* express the concept: "for all nodes, if a node (`n`) is in the list (`l`) then that node `n` is not reachable from itself." The fault is in red and reflects the incorrect use of reflexive transitive closure instead of transitive closure (^). Line 5 depicts an Alloy command that executes `Acyclic`. During execution, the Analyzer will search for instances, which are assignments to the sets and relations of the model such that all formulas invoked are true. This search is restricted to a scope, a user provided upper bound on the universe of discourse. The command on line 5 uses Alloy's default scope of 3, meaning that any satisfying instance will contain at most 3 `List` atoms and 3 `Node` atoms.

AUnit addresses the need to have a systematic method to verify the correctness of Alloy models. AUnit defines testing in Alloy's declarative environment – the SAT back-end looks for all satisfying instances in one execution – by answering: (1) what is a test case and (2) what test execution and outcomes are.Figure 2 depicts a fault revealing AUnit test case. AUnit test cases consists of two portions: (1) a *valuation* in which the user outlines an instance she wants to reason over (lines 1-4) and (2) a *command* that specifies the formulas under test (lines 5-6). A test *passes* if the valuation is a valid instance of the command.The test in Figure 2 fails because the incorrect use of reflexive transitive closure in Figure 1 builds a set that includes the node (`n`) itself. Therefore, `Node0` and `Node1` are incorrectly considered to be "reachable from themselves."

To use AlloyFL to localize the fault, we use an automatically generated test suite of 22 tests, including the test in Figure 2. We configure AlloyFL to use the Ochiai formula [1] and compute a weighted score using a ratio of 60% AlloyFL$_{co}$ score and 40% AlloyFL$_{mu}$score. The output of our execution is shown in Figure 3. In the left panel, suspicious locations for the displayed model are highlighted, with red indicating a highly suspicious location. In the right panel, a list of suspicious locations is depicted with supplementary information. The actual faulty expression "(*`link`)" located within the formula "`n !in n.*link`" is revealed to be the most suspicious location with a weighted suspiciousness score of 0.829. This fault further motivates the difficulty in localizing Alloy bugs where the difference between a correct expression and an incorrect one can be a single symbol contained within a larger expression or formula.

Table 1: Suspiciousness Formulas in AlloyFL.

Name	Formula
Tarantula [6]	$\frac{\frac{failed(e)}{totalfailed}}{\frac{failed(e)}{totalfailed}+\frac{passed(e)}{totalpassed}}$
Ochiai [1]	$\frac{failed(e)}{\sqrt{totalfailed \times (failed(e)+passed(e))}}$
Op2 [12]	$failed(e) - \frac{passed(e)}{totalpassed+1}$
Barinel [2]	$1 - \frac{passed(e)}{passed(e)+failed(e)}$
DStar [22]	$\frac{failed(e)^*}{passed(e)+(totalfailed-failed(e))}$
totalfailed: total number of tests that failed *totalpassed*: total number of tests that passed *failed(e)*: number of failed tests that cover or kill *e* *passed(e)*: number of passed tests that cover or kill *e*	

3 TECHNIQUE

AlloyFL determines which AST nodes of a model are most likely to be faulty. To achieve this, AlloyFL uses a suspiciousness formula and combines two different fault localization strategies.

3.1 Suspiciousness Formulas

AlloyFL supports five different suspiciousness formulas: (1) Tarantula [6], (2) Ochiai [1], (3) Op2 [12], (4) Barinel [2] and (5) DStar [22]. These formulas are outlined in Figure 1 and are commonly used for SBFL for imperative languages. For AlloyFL$_{co}$, the code elements (e) are AST nodes. For AlloyFL$_{mu}$, mutations of killed mutants are treated as covered code elements (e) while mutations of live mutants are treated as uncovered code elements. Since, each mutation is tied to the AST node that gets mutated, AlloyFL$_{mu}$ scores as still tied to AST nodes. *totalfailed* and *totalpassed* are the number of tests which failed and passed for the original model. *failed(e)* and *passed(e)* are the number of failing and passing tests that cover the AST node or kill the mutant *e*. In general, for AlloyFL$_{co}$, if a node is covered by more failing tests but fewer passing tests, then it is assigned a higher suspiciousness score. For AlloyFL$_{mu}$, if a mutated node makes more failing tests pass but fewer passing tests fail, then it is assigned a higher suspiciousness score.

3.2 Fault Localization Strategy

The AlloyFL extension implements AlloyFL$_{hy}$, a hybrid fault localization techniques that combines the results of AlloyFL$_{co}$ and AlloyFL$_{mu}$. We first describe how AlloyFL$_{co}$ and AlloyFL$_{mu}$ work, and then highlight how we combine the two approaches.

3.3 Spectrum-Based Fault Localization

In traditional imperative programs, fault localization will use control-flow and execution traces to implement spectrum-based fault localization techniques. In contrast, Alloy's declarative execution

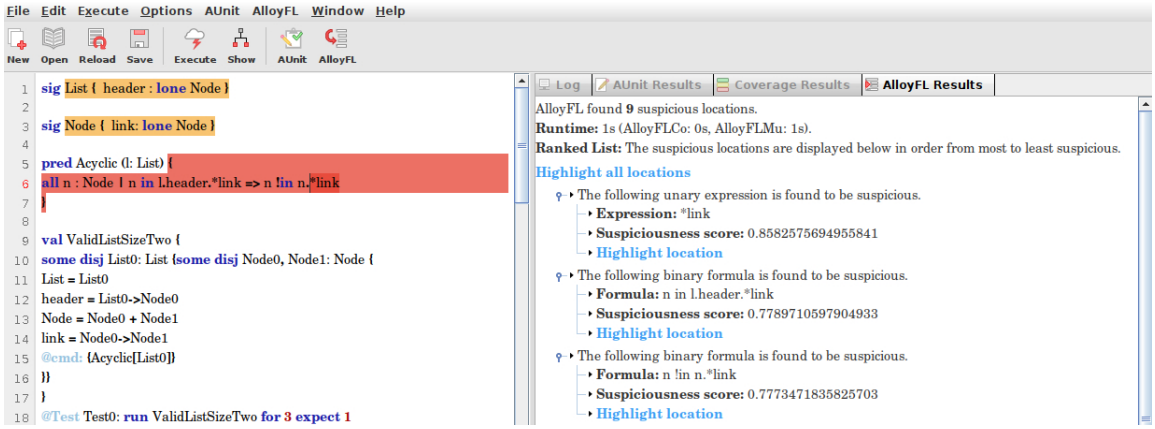

Figure 3: AlloyFL GUI Results

environment does not have control-flow. Instead, when a test is executed, every constraint in any invoked paragraph will be executed together. As a result, all AST nodes declared in the same paragraph share the same suspiciousness score. Therefore, AlloyFL$_{co}$ computes a suspiciousness score for each Alloy paragraph based on the number of passing/failing tests that cover it and a formula shown in Table 1. However, we are able to optimize this execution viewpoint by making use of a static analyzer that finds all Alloy paragraphs transitively used by a test, but it ignores dependencies that are never used. For example, if a test uses an expression "`all s: S, t: T | some s && p[s]`" where variable "`t`" is not used, then the test only depends on signature "`S`" and predicate "`p[...]`". Of note, for Alloy executions, all facts are implicitly used, and all paragraphs transitively invoked in the facts are covered by each test. As output, AlloyFL$_{co}$ produces a list of paragraphs ranked in descending order of suspiciousness score. In case of a tie, AlloyFL$_{co}$ prioritizes the paragraph with a smaller number of AST nodes.

3.4 Mutation-Based Fault Localization

To perform mutation-based fault localization, AlloyFL$_{mu}$ uses the set of mutation operators outlined by MuAlloy [17]. These operators mutate different nodes in the AST representation of the model and span the breadth of the Alloy grammar. To perform MBFL, AlloyFL$_{mu}$ collects all AST nodes covered by the failing tests. Then, AlloyFL$_{mu}$ iterates over each node n and applies all valid mutants to n. A mutant is considered valid if the mutated model does not result in a compilation error and the mutated model is not equivalent to the original model. Unlike imperative-based mutation testing, AlloyFL$_{mu}$ is able to use the Alloy language itself to check for equivalence between the original and mutated models with respect to a given bound. For every valid mutant of n, a suspiciousness score of that mutant is calculated by executing the original test suite over the mutated model and plugging the results into the user selected suspiciousness formula. After exploring all valid mutants of n, AlloyFL$_{mu}$ retains the largest score found for n. Once all AST nodes covered by failing tests are explored, AlloyFL$_{mu}$ returns the list of AST nodes sorted in descending order of suspiciousness scores. As with AlloyFL$_{co}$, in case of a tie, AlloyFL$_{mu}$ prioritizes the paragraph with a smaller number of AST nodes. Additionally, any AST node with a negative suspiciousness score is removed from the final list presented to the user.

3.5 Hybrid-Based Fault Localization

AlloyFL implements AlloyFL$_{hy}$, which is a hybrid technique that leverages both AlloyFL$_{co}$ and AlloyFL$_{mu}$. To achieve this, for a given AST node n, AlloyFL$_{hy}$ calculates a weighted score for n that combines n's AlloyFL$_{co}$ (S_{co}) and AlloyFL$_{mu}$ (S_{mu}) scores. Specifically, AlloyFL$_{hy}$ computes the weighted sum as $(1-\lambda)S_{co} + \lambda S_{mu}$, where $0 \leq \lambda \leq 1$. AlloyFL$_{hy}$ allows us to take advantage of the fact that AlloyFL$_{mu}$ and AlloyFL$_{co}$ have different strengths and weaknesses for localizing different types of faults. Most notably, AlloyFL$_{mu}$ can struggle to localize omission errors in which case AlloyFL$_{co}$ performs relatively well. Thus, AlloyFL$_{hy}$ benefits from both AlloyFL$_{co}$ and AlloyFL$_{mu}$.

4 ANALYZER INTEGRATION

AlloyFL is a self contained executable jar file written in Java. Our implementation of AlloyFL is built of top of the AUnit Analyzer [16], which is an extension to the latest stable release of the Analyzer [3] (version 5.0.1) that includes native support for AUnit. To support AlloyFL, the Analyzer is extended to allow the user to: (1) configure the AlloyFL execution and (2) update the visual feedback from the Analyzer to reflect AlloyFL's ranked list of suspicious locations.

The Analyzer is split into two main panels: (1) the left-hand panel is a text editor where users can create Alloy models and (2) the right-hand panel displays logging information. To use AlloyFL, the user first opens or builds a faulty Alloy model and a fault revealing AUnit test suite in the editor panel, as they would any other Alloy model. Then, the user can configure the AlloyFL execution. AlloyFL is packaged with the following default settings: the Ochiai suspiciousness formula and a weight of 0.4 (40% AlloyFL$_{mu}$ score and 60% AlloyFL$_{co}$ score). The AlloyFL menu options allow the user to change both. For the suspiciousness formula, the user is given a list comprised of all the formulas in Table 1. For the weight, uses can select a value between 0, which means only AlloyFL$_{co}$'s score will be used, and 1, which means only AlloyFL$_{mu}$ score will be used.

Table 2: AlloyFL Model Stats and Execution Results

Model	#AST	#Flt	#Test	Scp	Time (sec)	Rank
addr	124	1	30	3	6	1:19
array	68	2	23	3	4	1:7
bst	175	4	110	4	48	1:37
cd	52	2	25	3	1	2:18
ctree	76	1	22	3	3	9:17
dll	92	2	49	3	7	1:7
fsm	85	2	15	3	2	1:27
grade	77	1	41	3	2	1:10
other	40	1	21	3	1	5:11
scl	201	3	87	3	43	1:36

Additionally, to help with the adoption of AlloyFL, the menu also includes a prompt to view an AlloyFL tutorial.

The user can run AlloyFL either from the icon menu bar or the AlloyFL menu. Once AlloyFL successfully executes, the user is presented with the results in two key ways. First, all suspicious locations are highlighted in the text editor, ranging from deep red to indicate a highly suspicious location to light yellow indicate a slightly suspicious location. An example of this behavior can be seen in Figure 3. Second, AlloyFL generates a results tab which summarizes the execution by presenting the total number of suspicious AST nodes and presents a breakdown of the runtime. Additionally, the results tab displays a ranked list of suspicious locations from most to least suspicious. For each suspicious location, the user is shown the constraint, the suspiciousness score, and an interactive link which highlights the location in the editor pane. A portion of the ranked list output can be seen in Figure 3. This individual highlighting features helps clearly convey to user what specific portion of the model is being referenced, as a formula or expression may appear more than once in a model.

5 EVALUATION

We evaluate AlloyFL on a machine running Ubuntu 20.04 LTS with 1.8GHz Intel Core i7 CPU and 16 GB RAM. AlloyFL is set up to use the Ochiai formula and an impact weight of 0.4 [21].

5.1 Real-World Faulty Subjects

We present a small but representative evaluation of AlloyFL over 10 real world faulty models. Address book (**addr**) is from Alloy's example set, which was incorrect in earlier versions of Alloy. Grade book (**grade**) and other groups (**other**) are Alloy translations of access-control specifications used in Amalgam [14]. Colored tree (**ctree**) is from MuAlloy [17]. These four models represent faults introduced by more experienced Alloy users. Array (**array**), binary search tree (**bst**), class diagram (**cd**), doubly-linked list (**dll**), finite state machine (**fsm**), and singly-linked list with sorting and counting (**scl**) are homework questions we collected from graduate students, which reflect faults created by new Alloy users.

To convey complexity, we report four different metrics in Table 2 related to the size of the fault localization problem: (1) column two (**#AST**) show the total number of AST nodes in the model, (2) column three (**#Flt**) presents the number of faults in the model, (3) column four (**#Test**) depicts the size of the test suite and (4) column five (**Scp**) shows the maximum scope used to run the tests. Since many of these faulty models did not come with AUnit test suites, for our experiments, we automatically generated a test suite using MuAlloy, which has been shown to be effective at revealing faults in real world models [17, 20].

5.2 Results

To evaluate AlloyFL, we use the total execution time and the ranking of the actual faulty location to measure efficiency and effectiveness respectively. In Table 2 column five (**Time**) presents the runtime from the time the user presses the button to run AlloyFL to the time the results are presented to the user in seconds. AlloyFL's runtime does increase as both the number of AST nodes and the size of the test suite increases, both of which increase the size of the fault localization problem. For all executions, the AlloyFL$_{mu}$ portion of AlloyFL$_{hy}$ takes up a majority of the execution time. Since AlloyFL$_{mu}$ performs mutation testing, it is expected that AlloyFL$_{mu}$'s runtime would increase as the size of the test suite increases. However, the overhead of AlloyFL is not prohibitive as all models run in under a minute.

Column six (**Rank**) presents a ratio depicting the rank of the faulty location in the list reported to the user and the total number of suspicious locations. For example, for the model **scl**, the rank 1:36 means an actual faulty location was reported as the first suspicious location out of 36 total suspicious locations. If there is more than one faulty location, column six will reflect the highest ranked actually faulty node. For 7 of the 10 models, AlloyFL reports a faulty location as the highest suspicious node. Furthermore, for **cd**, the top two nodes have the same suspiciousness score; however, the real faulty location encapsulated a larger formula and was ranked second instead of first. While AlloyFL often reports a faulty location as the most suspicious location, AlloyFL can struggle with under-constrained faults (**ctree** and **other**), in which the error is the omission of a formula. For example, for **ctree**, while AlloyFL is able to flag the unconstrained fact `undirected` as suspicious, the location is ranked as 9th out of 17 locations. AlloyFL also works in the presence of multiple faults. To highlight AlloyFL's performance with multiple faults present, we can look at **bst**'s ranked list beyond just the top result. For **bst**, AlloyFL flags three of **bst**'s four faulty locations in the top 5 suspicious locations reported.

6 CONCLUSION

This paper introduces the open-source AlloyFL tool for fault localization of Alloy models. To localize a fault, AlloyFL uses an AUnit test suite with at least one failing test, a user-selected suspiciousness formula and a user selected weight, to create a ranked list of suspicious locations in the faulty Alloy model. In addition, as part of the reporting, suspicious portions of the model are highlighted from yellow to red, depending on their suspiciousness score. Experimental results reveal AlloyFL is effective at ranking faulty locations, works in the presence of multiple faults and localizes faults quickly. Since AlloyFL is packaged as an extension of Alloy's IDE, AlloyFL paves the way for new users to explore Alloy while benefiting form the use of AlloyFL in their development practices.

ACKNOWLEDGMENTS

This work is partially supported by the National Science Foundation under Grant No. CCF-2042871.

REFERENCES

[1] Rui Abreu, Peter Zoeteweij, Rob Golsteijn, and Arjan J. C. van Gemund. 2009. A Practical Evaluation of Spectrum-based Fault Localization. *JSS* (2009).

[2] Rui Abreu, Peter Zoeteweij, and Arjan JC Van Gemund. 2009. Spectrum-based multiple fault localization. In *ASE*.

[3] Alloy analyzer Website. 2019. http://alloytools.org. (2019).

[4] Daniel Jackson. 2002. Alloy: A Lightweight Object Modelling Notation. *TOSEM* (2002).

[5] Daniel Jackson. 2017. The Alloyed Joys of Software Engineering Research. http://people.csail.mit.edu/dnj/talks/icse17/icse17-nobuilds.pdf.

[6] James A. Jones and Mary Jean Harrold. 2005. Empirical Evaluation of the Tarantula Automatic Fault-localization Technique. In *ASE*.

[7] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. 1999. JML: A Notation for Detailed Design. In *Behavioral Specifications of Businesses and Systems*.

[8] Ferney A Maldonado-Lopez, Jaime Chavarriaga, and Yezid Donoso. 2014. Detecting network policy conflicts using Alloy. In *ABZ*.

[9] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. 2011. CD2Alloy: Class Diagrams Analysis Using Alloy Revisited. In *MODELS*.

[10] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. 2011. CDDiff: Semantic Differencing for Class Diagrams. In *ECOOP*.

[11] Darko Marinov and Sarfraz Khurshid. 2001. TestEra: A Novel Framework for Automated Testing of Java Programs. In *ASE*.

[12] Lee Naish, Hua Jie Lee, and Kotagiri Ramamohanarao. 2011. A Model for Spectra-based Software Diagnosis. *TSE* (2011).

[13] Timothy Nelson, Christopher Barratt, Daniel J. Dougherty, Kathi Fisler, and Shriram Krishnamurthi. 2010. The Margrave Tool for Firewall Analysis. In *LISA*.

[14] Tim Nelson, Natasha Danas, Daniel J. Dougherty, and Shriram Krishnamurthi. 2017. The Power of "Why" and "Why Not": Enriching Scenario Exploration with Provenance. In *FSE*.

[15] Natali Ruchansky and Davide Proserpio. 2013. A (Not) NICE Way to Verify the Openflow Switch Specification: Formal Modelling of the Openflow Switch Using Alloy. *SIGCOMM* (2013).

[16] Allison Sullivan, Kaiyuan Wang, and Sarfraz Khurshid. 2018. AUnit: A Test Automation Tool for Alloy. In *ICST*. 398–403.

[17] Allison Sullivan, Kaiyuan Wang, Razieh Nokhbeh Zaeem, and Sarfraz Khurshid. 2017. Automated Test Generation and Mutation Testing for Alloy. In *ICST*.

[18] Allison Sullivan, Razieh Nokhbeh Zaeem, Sarfraz Khurshid, and Darko Marinov. 2014. Towards a Test Automation Framework for Alloy. In *SPIN*.

[19] Caroline Trippel, Daniel Lustig, and Margaret Martonosi. 2019. Security Verification via Automatic Hardware-Aware Exploit Synthesis: The CheckMate Approach. *IEEE Micro* (2019).

[20] Kaiyuan Wang. 2015. *muAlloy – An Automated Mutation System for Alloy.* Master's thesis. University of Texas at Austin.

[21] Kaiyuan Wang, Allison Sullivan, and Sarfraz Khurshid. 2020. Fault Localization for Declarative Models in Alloy. In *ISSRE*.

[22] W. E. Wong, V. Debroy, R. Gao, and Y. Li. 2014. The DStar Method for Effective Software Fault Localization. *IEEE Transactions on Reliability* (2014).